ДОДАТОК А

Графічний матеріал кваліфікаційної роботи

Харківський національний університет радіоелектроніки
Кафедра ЕОМ

Розробка 2D гри на рушії Unity

Кваліфікаційна робота
Перший (Бакалаврський рівень)

Здобувач:
Вадим ПЕРЕПЕЛКО
КІУКІ-21-5

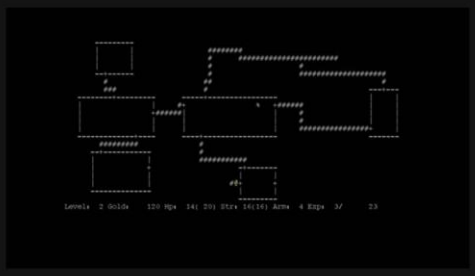
Керівник:
Володимир АРГУНОВ
асистент кафедри ЕОМ

Мета роботи

Метою кваліфікаційної роботи є розробка прототипу комп'ютерної відеогри з елементами жанру roguelike, використовуючи рушій Unity.
Проаналізувати розробку комп'ютерних ігор, жанр створюваної гри, схожі продукти, їх відмінності та особливості.

Жанр rogue-like

- Перманентна смерть
  після смерті прогрес скидається.
- Процедурна генерація
  для кожного проходження рівні генерується заново
- Варіативність
  наявність механік що спіткають гравця на перепроходження



Вигляд оригінальної гри "Rogue" (1980)

Аналіз сучасних проектів жанру

ENTER THE GUNGEON

Динамічний шутер у стилі bullet hell, де гравець досліджує підземелля. Гра вирізняється перекатами невразливості та великою кількістю зброї з унікальними ефектами. Геймплей вимагає високої майстерності, швидкої реакції та тактичного мислення, підтримуючи гравця у постійній напрузі.

THE BINDING OF ISSAC

Гравець досліджує підземелля, бореться з ворогами та збирає сотні унікальних предметів. Комбінації цих предметів кардинально змінюють геймплей, забезпечуючи глибоку варіативність і високу реіграбельність.

HADES

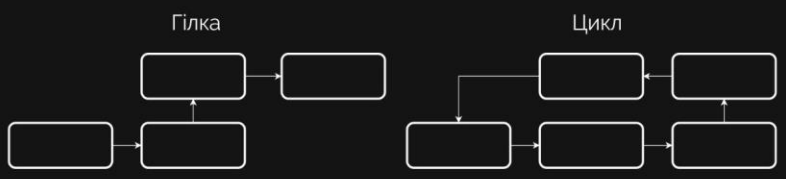
Roguelite-екшн у стилі грецької міфології, що поєднує швидкий бойовий процес із глибоким сюжетом. Завдяки дарам богів, системі покращень та наративній прогресії кожен забіг змінює як геймплей, так і розвиток історії.

Постановка завдання

Unity

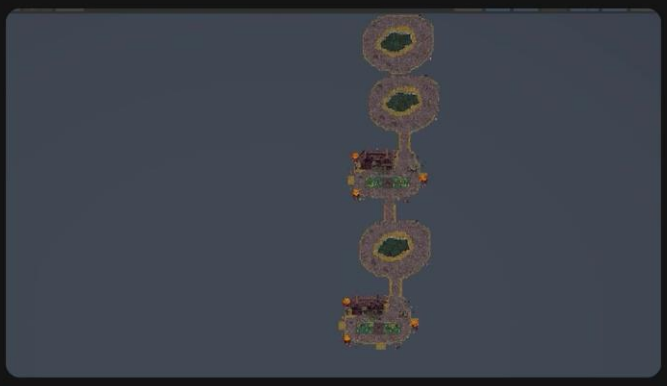
- Процедурна генерація світу
- Pixel art з видом зверху
- Бойова система
- Система предметів
- Вороги та система поведінок
- Інтерфейс

Процедурна генерація



Генератор вибирає один із доступних йому макетів рівня і на його основі починає генерацію.

Процедурна генерація



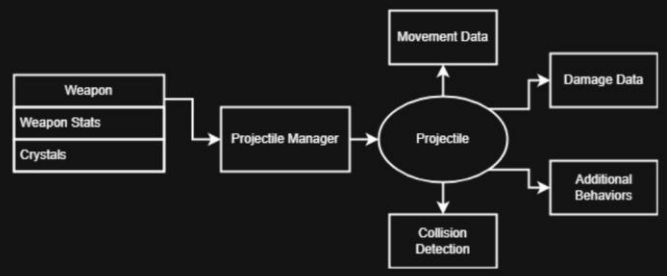
Система предметів та інвентар

Взаємодія із предметами на землі

Вигляд інвентарю гравця

Покупка предметів

Зброя та її параметри



ТИПИ ШКОДИ:

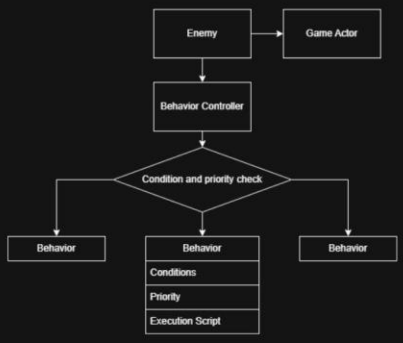
- ВОГОНЬ
- МАГІЯ
- ВОДА
- ОТРУТА
- ЕЛЕКТРИКА

Гравець здійснює постріл снарядом із магічною шкодою

Система ворогів

ТИПИ ПОВЕДІНОК:

- ПРОСТОЮВАННЯ
- ПЕРЕМІЩЕННЯ
- АТАКА



Ворог атакує гравця використовуючи одну зі своїх атак

Вороги під різними ефектами

Інтерфейс

Здоров'я гравця та кількість коштів

Міні-мапа

Туман війни

Існує для створення атмосфери, і для приховання невідкритих та недоступних частин гри.

Туман існує лише в межах камери гравця

Вигляд туману у грі

Висновки

- Розроблено функціональний прототип гри у жанрі Rogue-like з видос зверху
- Створено процедурно створюваний світ з різними кімнатами
- Систему предметів та їх взаємодії зі зброєю
- Систему ворогів
- Систему туману війни

Перспективи на розвиток:

- Додавання мета-прогресу
- Збільшення різноманітності ворогів та їх поведінок
- Більше рівнів
- Покращення та оптимізація роботи генерації

ДОДАТОК Б
КОД МОДУЛІВ ГРИ

Б.1 Посилання на проєкт:

https://drive.google.com/drive/folders/1TNBl7IqH7A8VTARQH-na1krHnOdTB9hu?usp=sharing

Б.2 Файли та методи

Б.2.1 Файл Exit.cs

```
using System;
using System.Collections.Generic;
using UnityEngine;

Serializable]
public class Exit
{
    [NonSerialized] public bool isFree = true;
    public Transform transform;
    public Vector2Int direction;
    [NonSerialized] public Room room;
    public Exit connectedExit;

    public List<GameObject> connectedHallway = new();

    public GameObject ToHide;

    public Vector2 WorldPosition => transform.position;

    public Vector2 LocalOffset => transform.position -
transform.root.position;


    public void Connect(Exit other)
    {
        isFree = false;
        connectedExit = other;
        if (ToHide != null) ToHide.gameObject.SetActive(false);
    }

    public void Disconnect()
    {
        isFree = true;
        connectedExit = null;
```

```
        connectedHallway.Clear();
        if (ToHide != null) ToHide.gameObject.SetActive(true);
    }

    public void ConnectBoth(Exit other, List<GameObject>
connectedHallway)
    {
        Connect(other);
        other.Connect(this);
        this.connectedHallway = new(connectedHallway);
        other.connectedHallway = this.connectedHallway;
    }
    public void DisconnectBoth()
    {
        connectedExit.Disconnect();
        Disconnect();
    }
}
```

Б.2.2 Метод TryPlaceRoomWithHallway()

```
protected IEnumerator TryPlaceRoomWithHallway(Room room, Room
previousRoom, Exit fromExit, List<GameObject> hallways,
Vector2Int direction, Action<bool> callback)
    {
        int length = 1;
        Vector3 start = fromExit.transform.position +
(Vector3)new(direction.x * 0.5f, direction.y * 0.5f);
        Room hallSegment = CreateHallway(start, direction);
        var boundsToIgnore = new List<GameObject> {
hallSegment.RoomCollider.gameObject,
room.RoomCollider.gameObject };
        var segments = new List<GameObject> {
hallSegment.gameObject };
        yield return null;
        while (CheckOverlap(room.RoomCollider, boundsToIgnore)
|| length < MinHallwayLength)
        {
            if (length >= MaxHallwayLength)
            {
                CleanupHallways(segments);
                callback(false);
                yield break;
            }

            Vector3 offset = new(direction.x / 2f, direction.y /
2f, 0);
            List<Room> localSegments = new();
            for (int i = 0; i < 10; i++)
            {
                room.transform.position += offset;
                start += offset;
                hallSegment = CreateHallway(start, direction);
```

```
boundsToIgnore.Add(hallSegment.RoomCollider.gameObject);
                segments.Add(hallSegment.gameObject);
                localSegments.Add(hallSegment);
                length++;
            }
            yield return null;

            var checkIgnore = new
List<GameObject>(boundsToIgnore) {
previousRoom.RoomCollider.gameObject };
            foreach (var localSegment in localSegments)
            {
                if (CheckOverlap(localSegment.RoomCollider,
checkIgnore))
                {
                    CleanupHallways(segments);
                    callback(false);
                    yield break;
                }
            }
            yield return null;
        }
        hallways.AddRange(segments);
        callback(true);
    }
```

Б.2.3 Файл DungeonGenerator.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using NavMeshPlus.Components;
using System;

public class DungeonGenerator : MonoBehaviour
{
    public List<DungeonScriptable> dungeonScriptables = new();

    public RoomPool RoomPool;
    [NonSerialized] public Vector2Int DungeonDirection;
    private DungeonScriptable currentDungeon;

    private Dictionary<int, LayoutGenerator> layoutInstances =
new();
    private Dictionary<int, GameObject> layoutRoots = new();

    [SerializeField] private Vector2 layoutTestOffset = new
Vector2(-9999, -9999);
    [SerializeField] private float layoutSpacing = 100; //
Minimum spacing when testing placement

    public NavMeshSurface surface;
```

```
    float timeToGen = 0f;

    private void Awake()
    {
        if (dungeonScriptables.Count <= 0) return;
        currentDungeon =
dungeonScriptables[UnityEngine.Random.Range(0,
dungeonScriptables.Count)];
        int rand = UnityEngine.Random.Range(1, 5);
        switch (rand)
        {
            case > 3:
                DungeonDirection = new(1, 0);
                break;
            case > 2:
                DungeonDirection = new(-1, 0);
                break;
            case > 1:
                DungeonDirection = new(0, 1);
                break;
            default:
                DungeonDirection = new(0, -1);
                break;

        }
        Debug.Log("Dungeon direction :" + DungeonDirection);
    }

    public IEnumerator StartGeneration()
    {
        timeToGen = Time.time;
        if (currentDungeon == null || currentDungeon.layouts ==
null || currentDungeon.layouts.Count == 0)
        {
            Debug.LogError("No valid dungeon scriptable or
layouts!");
            yield break;
        }

        // Generating layouts
        // Setup generators and layout roots
        List<IEnumerator> allSetups = new();

        for (int i = 0; i < currentDungeon.layouts.Count; i++)
        {
            DungeonLayout layout = currentDungeon.layouts[i];

            LayoutGenerator generator = layout.type switch
            {
                DungeonLayout.LayoutType.loop => new
LoopGenerator(this),
                DungeonLayout.LayoutType.branch => new
```

```
BranchGenerator(this),
                _ => null
            };

            if (generator == null)
            {
                Debug.LogWarning($"Unknown layout type at index
{i}.");
                continue;
            }

            generator.roomPool = RoomPool;
            layoutInstances[layout.id] = generator;

            // Create layout root
            GameObject layoutRoot = new
GameObject($"LayoutRoot_{layout.id}");
            layoutRoot.transform.position = layoutTestOffset +
new Vector2(layoutSpacing * i, 0);
            layoutRoots[layout.id] = layoutRoot;

            // Room creation
            List<Room> rooms = new();
            foreach (RoomType roomType in layout.Rooms)
            {
                Room prefab = RoomPool.GetRoom(roomType);
                if (prefab != null)
                {
                    Room instance = Instantiate(prefab,
layoutRoot.transform);
                    instance.transform.localPosition =
Vector3.zero;
                    instance.name = $"{roomType}_{layout.id}";
                    rooms.Add(instance);
                }
            }

            // Add setup coroutine to list (not yield yet)
            allSetups.Add(generator.Setup(rooms,
layoutRoot.transform, DungeonDirection));
        }



        bool isGenerated = false;
        while (!isGenerated)
        {

            yield return RunAllSetups(allSetups); // Wait for
ALL setup coroutines to finish
            // Place and connect layouts
            for (int i = 0; i < currentDungeon.layouts.Count;
i++)
```

```
            {
                DungeonLayout layout =
currentDungeon.layouts[i];
                LayoutGenerator generator =
layoutInstances[layout.id];
                GameObject root = layoutRoots[layout.id];
                if (i == 0)
                {
                    root.transform.position = Vector3.zero;
                    continue;
                }

                // Get layout to connect to
                if
(!layoutInstances.TryGetValue(layout.connectToId, out var
targetLayout) ||
                    !layoutRoots.TryGetValue(layout.connectToId,
out var targetRoot))
                {
                    Debug.LogError($"Could not find layout to
connect to: ID {layout.connectToId}");
                    continue;
                }

                Room fromRoom = GetConnectionRoom(targetLayout,
layout.connectionType);
                Room toRoom = generator.GetFirstRoom();
                if (fromRoom == null || toRoom == null)
                {
                    Debug.LogWarning($"Missing connection rooms
for layout {layout.id}");
                    continue;
                }

                // Try multiple positions to find a collision-
free placement
                bool placed = false;
                const int maxAttempts = 30;

                List<Exit> ignores = new();

                for (int attempt = 0; attempt < maxAttempts &&
!placed; attempt++)
                {

                    Exit fromExit =
fromRoom.GetRandomExit(ignores);
                    if (fromExit == null) break;

                    ignores.Add(fromExit);

                    List<Exit> toExits =
toRoom.GetOppositeExits(fromExit);
```

```
                    if (toExits.Count == 0) continue;


                    foreach (var toExit in toExits)
                    {
                        yield return null;
                        Vector2 targetExitWorldPos =
fromExit.WorldPosition;
                        Vector2 localOffset =
toExit.LocalOffset;
                        Vector3 desiredRootPosition =
targetExitWorldPos - localOffset;

                        Vector2Int direction =
fromExit.direction;

                        Vector3 start = targetExitWorldPos +
(Vector2)new(direction.x * 0.5f, direction.y * 0.5f);
                        Vector3 offset = new(direction.x / 2f,
direction.y / 2f);

                        generator.root.position =
desiredRootPosition + offset;


                        var segments = new List<GameObject>();


                        for (int j = 0; j < 4; j++)
                        {
                            var boundsToIgnore = new
List<GameObject>();
                            for (int h = 0; h < 10; h++)
                            {
                                Room hallSegment =
targetLayout.CreateHallway(start, direction);


segments.Add(hallSegment.gameObject);
                                if (h == 9)
boundsToIgnore.Add(hallSegment.RoomCollider.gameObject);

                                generator.root.position +=
offset;
                                start += offset;
                            }

                            yield return null;
                            LayoutGenerator.CheckLayoutCallback
callback = generator.CheckLayoutOverlap(boundsToIgnore);
                            if (callback ==
LayoutGenerator.CheckLayoutCallback.hallwayCollide)
                            {
```

```
                                break;
                            }

                            if (callback ==
LayoutGenerator.CheckLayoutCallback.none)
                            {
                                placed = true;
                                fromExit.ConnectBoth(toExit,
segments);
                                break;
                            }
                        }
                        if (placed)
                        {
                            break;
                        }

                        targetLayout.CleanupHallways(segments);
                    }
                }

                if (!placed)
                {
                    int rand = UnityEngine.Random.Range(1, 5);
                    Vector2Int oldDirection = DungeonDirection;
                    switch (rand)
                    {
                        case > 3:
                            DungeonDirection = new(1, 0);
                            break;
                        case > 2:
                            DungeonDirection = new(-1, 0);
                            break;
                        case > 1:
                            DungeonDirection = new(0, 1);
                            break;
                        default:
                            DungeonDirection = new(0, -1);
                            break;
                    }
                    if (DungeonDirection == oldDirection)
DungeonDirection = new(DungeonDirection.x * -1,
DungeonDirection.y * -1);

                    allSetups.Clear();
                    foreach (var pair in layoutInstances)
                    {
                        var entry = pair.Value;
                        var rooms = new
List<Room>(entry.PlacedRooms);
                        entry.Reset();

                        allSetups.Add(entry.Setup(rooms,
```

```
layoutRoots[pair.Key].transform, DungeonDirection));

                    }
                    break;
                }
                else isGenerated = true;
            }
        }
        timeToGen = Time.time - timeToGen;
        yield return OnGenerationComplete();
    }
    private IEnumerator OnGenerationComplete()
    {
        Debug.Log($"Dungeon generation finished in{timeToGen}");

        foreach (var entry in layoutInstances)
        {
            entry.Value.OnGenerationComplete();
        }
        yield return null;
    }
    private Room GetConnectionRoom(LayoutGenerator generator,
DungeonLayout.LayoutConnectionType connectionType)
    {
        return connectionType switch
        {
            DungeonLayout.LayoutConnectionType.first =>
generator.GetFirstRoom(),
            DungeonLayout.LayoutConnectionType.last =>
generator.GetLastRoom(),
            DungeonLayout.LayoutConnectionType.random =>
generator.GetRandomRoom(),
            _ => generator.GetLastRoom()
        };
    }
    private IEnumerator RunAllSetups(List<IEnumerator> setups)
    {
        int doneCount = 0;
        int total = setups.Count;

        foreach (var setup in setups)
        {
            StartCoroutine(WrappedSetup(setup, () =>
doneCount++));
        }

        while (doneCount < total)
        {
            yield return null;
        }
    }

    private IEnumerator WrappedSetup(IEnumerator coroutine,
```

```
System.Action onComplete)
    {
        yield return coroutine;
        onComplete?.Invoke();
    }
}
```

Б.2.4 Файл LoopGenerator.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class LoopGenerator : LayoutGenerator
{
    public LoopGenerator(DungeonGenerator dungeonGenerator) :
base(dungeonGenerator)
    {
    }

    public override IEnumerator Setup(List<Room> rooms,
Transform startTransform, Vector2Int genDirection)
    {
        Room startRoom = rooms[0];

        this.root = startTransform;
        this.genDireciton = genDirection;

        startRoom.transform.position = startTransform.position;

        yield return GenerateLoopCoroutine(rooms, startRoom,
false);
    }
    public override IEnumerator Setup(List<Room> rooms, Room
startRoom, Vector2Int genDirection)
    {
        this.root = startRoom.transform;
        this.genDireciton = genDirection;

        yield return GenerateLoopCoroutine(rooms, startRoom,
true);
    }

    private IEnumerator GenerateLoopCoroutine(List<Room> rooms,
Room startRoom, bool isStartRoomOnOtherLayout)
    {
        if (!isStartRoomOnOtherLayout)
PlacedRooms.Add(startRoom);


        Room prevRoom = startRoom;
```

```
        Exit prevExit =
startRoom.GetExitByDirection(genDireciton);

        List<Exit> ignores = new();

        for (int i = isStartRoomOnOtherLayout ? 0 : 1; i <
rooms.Count; i++)
        {
            Room current = rooms[i];
            bool placed = false;

            Exit fromExit =
prevRoom.GetExitByDirection(genDireciton);
            Exit toExit = current.GetOppositeExit(fromExit);

            for (int tries = 0; tries < MaxTries && !placed;
tries++)
            {
                ignores.Add(fromExit);
                PositionRoomToConnect(current, toExit,
fromExit);
                yield return null;

                bool success = false;
                var hallways = new List<GameObject>();
                yield return TryPlaceRoomWithHallway(current,
prevRoom, fromExit, hallways, fromExit.direction, r => success =
r);

                if (success)
                {
                    fromExit.ConnectBoth(toExit, hallways);
                    PlacedRooms.Add(current);
                    prevRoom = current;
                    prevExit = toExit;
                    placed = true;
                }
                else
                {
                    Exit newExit =
prevRoom.GetExitByDirection(genDireciton, ignores);
                    if (newExit == null) newExit =
prevRoom.GetExitByRelativeDirection(genDireciton, ignores);
                    if (newExit == null)
prevRoom.GetRandomExit(ignores);

                    fromExit = newExit;
                    toExit = current.GetOppositeExit(fromExit);
                }
            }

            if (!placed)
            {
```

```
                //rooms[i] = roomPool.GetReplacement(current);
                //i--;
            }
        }
        bool isReconected = false;
        yield return ReconnectLoopToStart(genDireciton, r =>
isReconected = r);
    }

    private IEnumerator ReconnectLoopToStart(Vector2Int
startDireciton, Action<bool> callback)
    {
        for (int i = PlacedRooms.Count - 1; i > 0; i--)
        {
            Room current = PlacedRooms[i];
            Room next = i == PlacedRooms.Count - 1 ?
PlacedRooms[0] : PlacedRooms[i + 1];
            Room before = PlacedRooms[i - 1];

            if (i < PlacedRooms.Count - 1 &&
next.DistanceToRoom(PlacedRooms[0]) >
current.DistanceToRoom(PlacedRooms[0]))
            {
                Room lastRoom = next;
                Room firstRoom = current;

                List<Exit> triedExitsA = new();
                List<Exit> triedExitsB = new();

                Exit fromExit =
lastRoom.GetExitByDirection(startDireciton);
                Exit toExit =
firstRoom.GetExitByDirection(startDireciton);

                List<Exit> ignores = new() { fromExit, toExit };

                for (int tries = 0; tries < MaxTries; tries++)
                {


                    if (fromExit == null || toExit == null)
yield break;

                    triedExitsA.Add(fromExit);
                    triedExitsB.Add(toExit);

                    bool connected = false;
                    var hallways = new List<GameObject>();
                    yield return
TryConnectWithSmartHallway(fromExit, toExit, hallways, result =>
connected = result);

                    if (connected)
```

```
                    {
                        fromExit.ConnectBoth(toExit, hallways);
                        yield break;
                    }
                    else
                    {
                        fromExit =
lastRoom.GetExitByDirection(startDireciton, ignores);
                        toExit =
firstRoom.GetExitByDirection(startDireciton, ignores);

                        if (fromExit == null)
                        ignores.Add(fromExit);
                        ignores.Add(toExit);

                        CleanupHallways(hallways);
                    }
                }
            }

            Exit toRemove = current.GetOccupiedExit();

            if (toRemove != null)
            {
                CleanupHallways(toRemove.connectedHallway);
                toRemove.DisconnectBoth();
            }

            bool placed = false;
            List<Exit> ignoreExits = new();

            for (int tries = 0; tries < MaxTries && !placed;
tries++)
            {
                Exit fromExit =
next.GetClosestExit(before.transform, ignoreExits);

                if (fromExit == null)
                {
                    callback(false);
                    yield break;
                }

                Exit toExit = current.GetOppositeExit(fromExit);



                ignoreExits.Add(fromExit);
                PositionRoomToConnect(current, toExit,
fromExit);

                bool success = false;
                var hallways = new List<GameObject>();
```

```
                yield return TryPlaceRoomWithHallway(current,
next, fromExit, hallways, fromExit.direction, r => success = r);

                if (success)
                {
                    fromExit.ConnectBoth(toExit, hallways);
                    placed = true;
                }
            }

            if (!placed)
            {
                PlacedRooms[i] =
roomPool.GetReplacement(current);
                i++;
            }
        }
    }
}
```

Б.2.5 Файл BranchGenerator.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BranchGenerator : LayoutGenerator
{
    public BranchGenerator(DungeonGenerator dungeonGenerator) :
base(dungeonGenerator)
    {
    }

    public override IEnumerator Setup(List<Room> rooms,
Transform startTransform, Vector2Int genDirection)
    {
        this.root = startTransform;
        this.genDireciton = genDirection;

        Room startRoom = rooms[0];
        startRoom.transform.position = startTransform.position;
        PlacedRooms.Add(startRoom);

        yield return GenerateBranchCoroutine(rooms, startRoom);
    }

    public override IEnumerator Setup(List<Room> rooms, Room
startRoom, Vector2Int genDirection)
    {
        yield return null;
    }
```

```
    private IEnumerator GenerateBranchCoroutine(List<Room>
rooms,Room startRoom)
    {

        Room prevRoom = startRoom;

        List<Exit> ignores = new();

        for (int i = 1; i < rooms.Count; i++)
        {
            Room current = rooms[i];
            bool placed = false;

            Exit fromExit = i == 1 ?
prevRoom.GetExitByDirection(genDireciton) :
prevRoom.GetExitByRelativeDirection(genDireciton);
            Exit toExit = current.GetOppositeExit(fromExit);

            for (int tries = 0; tries < MaxTries && !placed;
tries++)
            {
                ignores.Add(fromExit);

                PositionRoomToConnect(current, toExit,
fromExit);
                yield return null;

                bool success = false;
                var hallways = new List<GameObject>();
                yield return TryPlaceRoomWithHallway(current,
prevRoom, fromExit, hallways, fromExit.direction, r => success =
r);

                if (success)
                {
                    fromExit.ConnectBoth(toExit, hallways);
                    PlacedRooms.Add(current);
                    prevRoom = current;
                    placed = true;
                }
                else
                {
                    Exit newExit =
prevRoom.GetExitByRelativeDirection(genDireciton, ignores);
                    if (newExit == null) newExit =
prevRoom.GetRandomExit(ignores);

                    fromExit = newExit;
                    toExit = current.GetOppositeExit(fromExit);
                }
            }

            if (!placed)
```

```
            {
                //rooms[i] = roomPool.GetReplacement(current);
                //i--;
            }
        }
        yield break;
    }
}
```

Б.2.6 Файл Crystal.cs

```
using System.Collections.Generic;
using UnityEngine;

public class Crystal : Item
{
    [Header("Crystal Data")]
    public DamageType DamageType;
    public Color color;
    public List<StaffStatAppliable> StaffStats = new();
    public Slot CurrentStaffSlot;
    public Staff Staff;
    public SlotType ActiveSlotType = SlotType.back;

    protected override void Awake()
    {
        base.Awake();

        UIItem.Image.sprite = spriteRenderer.sprite;
        UIItem.Image.color = Color.white;
    }

    public void AssignToSlot(Slot slot)
    {
        CurrentStaffSlot = slot;
        Staff = slot.Staff;
        switch (slot.Type)
        {
            case SlotType.front :
            {
                OnAssignFront();
                break;
            }
            case SlotType.middle :
            {
                OnAssignMiddle();
                break;
            }
            case SlotType.back :
            {
                OnAssignBack();
                break;
            }
```

```
        }
    }

    public void RemoveFromSlot()
    {
        switch (CurrentStaffSlot.Type)
        {
            case SlotType.front :
            {
                OnRemoveFront();
                break;
            }
            case SlotType.middle :
            {
                OnRemoveMiddle();
                break;
            }
            case SlotType.back :
            {
                OnRemoveBack();
                break;
            }
        }

        Staff = null;
        CurrentStaffSlot = null;
    }

    protected virtual void OnAssignFront()
    {
        Staff.StaffStats.AddDamageType(DamageType);
    }

    protected virtual void OnAssignMiddle()
    {
        Staff.StaffStats.AddStats(StaffStats);
    }

    protected virtual void OnAssignBack()
    {
    }

    protected virtual void OnRemoveFront()
    {
        Staff.StaffStats.RemoveDamageType(DamageType);
    }

    protected virtual void OnRemoveMiddle()
    {
        Staff.StaffStats.RemoveStats(StaffStats);
    }

    protected virtual void OnRemoveBack()
```

```
    {
    }

    public void UseActive()
    {
        if (CurrentStaffSlot.Type != ActiveSlotType) return;

    }
}
```

Б.2.7 Файл Staff.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Staff : Item
{

    [Header("Projectile Data")]
    public string ProjectileBankName;
    public Projectile OverrideProjectile;

    [Space(10)]
    [Header("Crystal Slots Settings")]
    public GameObject SlotsUI;
    public List<Slot> _slots = new();


    [Space(10)]
    [Header("Shoot Settings")]
    public Transform ShootTransform;

    [Space(10)]
    [Header("Stats")]
    public StaffStats StaffStats = new();

    private new Collider2D collider;
    protected override void Awake()
    {
        base.Awake();
        StaffStats.Init();
        foreach (var entry in _slots)
        {
            entry.Init(this);
        }

        damageModel.TypeCondition =
StaffStats.ActiveDamageTypes;
        collider = GetComponent<Collider2D>();
    }
```

```
    public void OnCrystalAssigned(Slot _slot, Crystal
oldCrystal, Crystal newCrystal)
    {
        if(_slot == null || !_slots.Contains(_slot)) return;

        newCrystal?.AssignToSlot(_slot);
        oldCrystal?.RemoveFromSlot();

        damageModel.TypeCondition =
StaffStats.ActiveDamageTypes;
    }

    public void UseActive()
    {
        foreach (var slot in _slots)
        {
            slot.AssignedCrystal?.UseActive();
        }
    }


    public override void OnPickUp()
    {
        gameObject.SetActive(false);
        UIItem.gameObject.SetActive(true);

SlotsUI.transform.SetParent(Inventory.Parent_MainCrystal);
        collider.enabled = false;
    }

    public override void OnDrop()
    {
        gameObject.SetActive(true);
        UIItem.gameObject.SetActive(false);
        SlotsUI.transform.SetParent(transform);
        collider.enabled = true;
    }

}
```

Б.2.8 Файл StaffStat.cs

```
using System;
using UnityEngine;

public enum ApplyType
{
    ADDITIVE,
    MULTIPLICATIVE
}

[Serializable]
```

```
public class StaffStat
{
    public enum StatType
    {
        Damage,
        Speed,
        Bursts,
        PerBurst,
        BurstDelay,
        ShootDelay,
        ReloadTime
    }

    [NonSerialized] public StatType Type;
    [NonSerialized] public float Multiplier = 1;
    [NonSerialized] public float BaseValue;
    [SerializeField] private float _baseValue;

    public float Value { get => BaseValue * Multiplier; }

    public StaffStat(StatType type)
    {
        Type = type;
        Reset();
    }

    public void Reset()
    {
        BaseValue = _baseValue;
        Multiplier = 1f;
    }
}

[Serializable]
public struct StaffStatAppliable
{
    public StaffStat.StatType statType;
    public float value;
    public ApplyType applyType;
}
```

Б.2.9 Файл Interactable.cs

```
using UnityEngine;

[RequireComponent(typeof(Collider2D))]
public class Interactable : BaseBehaviour
{
    protected void Awake()
    {
        gameObject.layer = 7;
    }
```

```
    public virtual void Interact(Interactor interactor)
    {
        Debug.Log("Interacting with " + name);
    }

    public virtual void Approach(Interactor interactor)
    {
        Debug.Log("Approaching " + name);
    }

    public virtual void Depart(Interactor interactor)
    {
        Debug.Log("Departing from " + name);
    }
}
```

Б.2.10 Файл ActorDamageController.cs

```
using System.Collections;
using UnityEngine;

public class ActorDamageController : ProjectileCollider
{
    public float Health = 6;
    public float MaxHealth = 6;
    public float BlueHealth = 0;
    public float damageDelay = 1f;
    public bool invincible = false;
    public bool flashOnHit = true;

    private bool onDamageDelay = false;
    private Color flashColor = new(1, 1, 1, 0.2f);
    private float flashDuration = .1f;
    private Color originalColor;

    public override void OnProjectileCollision(Projectile
projectile)
    {
        if (onDamageDelay || bulletManager ==
projectile.BulletManager || projectile.damagesEnemies &&
gameActor.isPlayer || projectile.damagesPlayer &&
!gameActor.isPlayer) return;

        Damage(projectile.Damage, projectile._damageType);
        projectile.Die();
    }

    public void Damage(float damage, DamageType damageType)
    {
        if (onDamageDelay || invincible) return;


        if
```

```
(gameActor.Stats.DamageMultipliers.ContainsKey(damageType))
        {
            switch
(gameActor.Stats.DamageMultipliers[damageType])
            {
                case <= 0:
                    break;
                case < 1:
                    damage *=
gameActor.Stats.DamageMultipliers[damageType];
                    break;
                default:

gameActor.actorDamageEffects.Enable(damageType);
                    damage *=
gameActor.Stats.DamageMultipliers[damageType];
                    break;
            }
        }
        else
        {
            gameActor.actorDamageEffects.Enable(damageType);
        }

        Health -= damage;

        if (Health <= 0)
        {
            Die();
            return;
        }

        StartCoroutine(Flash());

        if (damageDelay > 0f) StartCoroutine(DamageDelay());
    }

    public void Damage(float damage)
    {
        if (onDamageDelay || invincible) return;
        Health -= damage;
        StartCoroutine(Flash());

        if (Health <= 0)
        {
            Die();
            return;
        }

        if (damageDelay > 0f) StartCoroutine(DamageDelay());
        else { animator.SetBool("OnDamaged", true); }
    }
```

```
    public void Damage(float damage, Color flashColor)
    {
        if (onDamageDelay || invincible) return;
        Health -= damage;
        StartCoroutine(Flash(flashColor));

        if (Health <= 0)
        {
            Die();
            return;
        }
        if (damageDelay > 0f) StartCoroutine(DamageDelay());
            else { animator.SetBool("OnDamaged", true); }
    }

    private IEnumerator DamageDelay()
    {
        onDamageDelay = true;
        animator.SetBool("OnDamaged", true);

        yield return new WaitForSeconds(damageDelay);

        onDamageDelay = false;
        animator.SetBool("OnDamaged", false);
        yield break;
    }

    private IEnumerator Flash()
    {
        if (originalColor == null || originalColor == default)
            originalColor =
spriteRenderer.material.GetColor("_Color_1");

        Color startColor = originalColor;
        Color targetColor = flashColor;
        targetColor.a = .4f;

        float halfDuration = flashDuration / 2f;

        // Fade to full alpha
        for (float t = 0; t < halfDuration; t += Time.deltaTime)
        {
            float lerpFactor = t / halfDuration;
            Color lerped = Color.Lerp(startColor, targetColor,
lerpFactor);
            spriteRenderer.material.SetColor("_Color_1",
lerped);
            yield return null;
        }
        spriteRenderer.material.SetColor("_Color_1",
targetColor);

        // Fade back to original alpha
```

```
        for (float t = 0; t < halfDuration; t += Time.deltaTime)
        {
            float lerpFactor = t / halfDuration;
            Color lerped = Color.Lerp(targetColor, startColor,
lerpFactor);
            spriteRenderer.material.SetColor("_Color_1",
lerped);
            yield return null;
        }

        spriteRenderer.material.SetColor("_Color_1",
originalColor);
    }

    public virtual void Die()
    {
    }

    private IEnumerator Flash(Color color)
    {
        if (originalColor == null) originalColor =
spriteRenderer.material.GetColor("_Color_1");

        spriteRenderer.material.SetColor("_Color_1", color ==
null ? flashColor : color);
        yield return new WaitForSeconds(flashDuration);
        spriteRenderer.material.SetColor("_Color_1",
originalColor);
    }
}
```

Б.2.11 Файл DamageModel.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DamageModel : BaseBehaviour
{
    [SerializeField]
    private List<DamageModelObject> m_states = new();

    public DamageType TypeCondition
    {
        get => _typeCondition;
        set
        {
            _typeCondition = value;
            OnConditionChange();
        }
    }
    public DamageType _typeCondition = DamageType.none;
```

```
    [SerializeField] public List<ParticleSystem> particleSystems
= new();


    [NonSerialized]
    public Sprite baseSprite;
    [NonSerialized]
    public Color baseColor;

    public void Awake()
    {
        if (!baseSprite) baseSprite = spriteRenderer.sprite;
        if (spriteRenderer.material.name ==
"Mat_Overlays")baseColor =
spriteRenderer.material.GetColor("_Color_2");
    }

    private void Reset()
    {
        spriteRenderer.sprite = baseSprite;
        spriteRenderer.material.SetColor("_Color_2", baseColor);

        foreach (var model in m_states)
        {
            if (model.Obj != null) model.Obj.SetActive(false);
        }
    }

    private void OnConditionChange()
    {
        Reset();

        Sprite targetSprite = baseSprite;
        Color targetColor = baseColor;

        foreach (var model in m_states)
        {
            if (_typeCondition == model.DamageTypeCondition)
            {

                if (model.Obj != null)
model.Obj.SetActive(true);
                targetSprite = model.Sprite != null ?
model.Sprite : targetSprite;
                targetColor = model.SpriteColor.a != 0 ?
model.SpriteColor : targetColor;
            }
        }

        spriteRenderer.sprite = targetSprite;
        spriteRenderer.material.SetColor("_Color_2",
targetColor);
```

```
    }


    public void OnDie()
    {
        if (particleSystems == null || particleSystems.Count <=
0) return;
        foreach (var system in particleSystems)
        {
            system.transform.parent = null;
            system.Stop();
            Destroy(system.gameObject, system.main.duration +
system.main.startLifetime.constantMax);
        }
    }

    [Serializable]
    public class DamageModelObject
    {
        public DamageType DamageTypeCondition;
        public GameObject Obj;
        public Color SpriteColor = Color.white;
        public Sprite Sprite;
    }
}
```

Б.2.12 Файл UIItem.cs

```
using System;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;

public class UIItem : MonoBehaviour, IBeginDragHandler,
IDragHandler, IEndDragHandler, IPointerEnterHandler,
IPointerExitHandler
{
    public Item Item;
    public ItemType Type
    {
        get => Item.Type;
        set => Item.Type = value;
    }
    public InventorySlot CurrentSlot;
    public Image Image;
    public Inventory Inventory
    {
        get => Item.Inventory;
    }
    [NonSerialized] public RectTransform RectTransform;
    private Vector3 m_originalScale;
    private Vector2 m_originalSize;
```

```
    private void Start()
    {
        RectTransform = GetComponent<RectTransform>();
        if (Image == null) Image = GetComponent<Image>();
        m_originalScale = RectTransform.localScale;
        m_originalSize = RectTransform.sizeDelta;
    }

    public void OnBeginDrag(PointerEventData eventData)
    {
        Inventory.ToolTipPanel.Hide();
        Inventory.ToolTipPanel.ForceHide = true;

        transform.position = Input.mousePosition;
        transform.SetParent(Inventory.Parent_Floating);

        RectTransform.localScale = new(m_originalScale.x * 1.3f,
m_originalScale.y * 1.3f);
        RectTransform.sizeDelta = m_originalSize;

        CurrentSlot.AssignedItem = null;

        Image.raycastTarget = false;
    }

    public void OnDrag(PointerEventData eventData)
    {
        transform.position = Input.mousePosition;
    }

    public void OnEndDrag(PointerEventData eventData)
    {
        Inventory.ToolTipPanel.ForceHide = false;
        RectTransform.localScale = m_originalScale;

        if (eventData.pointerCurrentRaycast.gameObject != null)
        {
            if (eventData.pointerCurrentRaycast.gameObject.tag
== "DropBox")
            {
                Inventory.DropItem(Item);
                CurrentSlot.AssignedItem = null;
                transform.SetParent(Item.transform);
                Image.raycastTarget = true;
                return;
            }

            if
(eventData.pointerCurrentRaycast.gameObject.TryGetComponent(out
InventorySlot _slot) &&
eventData.pointerCurrentRaycast.gameObject !=
CurrentSlot.gameObject)
            {
```

```
                if (_slot.AllowedTypes.Contains(Type))
                {
                    CurrentSlot.AssignedItem = null;
                    CurrentSlot = _slot;

                    transform.SetParent(CurrentSlot.transform);
                    _slot.AssignedItem = this;
                }
                else
                {
                    transform.SetParent(CurrentSlot.transform);
                    CurrentSlot.AssignedItem = this;
                }
            }
            else if
(eventData.pointerCurrentRaycast.gameObject.TryGetComponent(out
UIItem otherItem) && otherItem.Type == Type)
            {
                otherItem.transform.position =
CurrentSlot.transform.position;
                otherItem.transform.rotation =
Quaternion.Euler(0f,0f,CurrentSlot.SlotRotation);


otherItem.transform.SetParent(CurrentSlot.transform);

transform.SetParent(otherItem.CurrentSlot.transform);

                otherItem.CurrentSlot.AssignedItem = this;
                CurrentSlot.AssignedItem = otherItem;

                (otherItem.CurrentSlot, CurrentSlot) =
(CurrentSlot, otherItem.CurrentSlot);
            }
            else
            {
                transform.SetParent(CurrentSlot.transform);
                CurrentSlot.AssignedItem = this;
            }
        }
        else
        {
            transform.SetParent(CurrentSlot.transform);
            CurrentSlot.AssignedItem = this;
        }

        transform.position = CurrentSlot.transform.position;
        transform.rotation =
Quaternion.Euler(0f,0f,CurrentSlot.SlotRotation);

        Image.raycastTarget = true;
    }
```

```
    public void AssignToSlot(InventorySlot _slot)
    {
        if (CurrentSlot != null) CurrentSlot.AssignedItem =
null;

        CurrentSlot = _slot;

        transform.SetParent(CurrentSlot.transform);
        transform.position = CurrentSlot.transform.position;
        _slot.AssignedItem = this;
    }

    public void OnPointerEnter(PointerEventData eventData)
    {
        if (Inventory.ToolTipPanel == null) return;

        Inventory.ToolTipPanel.Show(Item.ToolTip);
    }

    public void OnPointerExit(PointerEventData eventData)
    {
        if (Inventory.ToolTipPanel == null) return;
        Inventory.ToolTipPanel.Hide();
    }
}
```

Б.2.13 Файл Inventory.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class Inventory : BaseBehaviour
{
    [SerializeField] private Animator m_InventoryAnimator;
    [Header("[== Items Data ==]")]
    [SerializeField] private List<Item> m_initItems = new();
    [NonSerialized] public List<Item> Items = new();

    [Header("[== Slots Data ==]")]
    [Header("Crystals")]
    [SerializeField] private InventorySlotUI
m_slotPrefabCrystal;
    [SerializeField] private GameObject m_crystalGrid;
    [SerializeField] private int m_crystalWidth = 6;
    [SerializeField] private int m_crystalHeight = 6;
    public List<InventorySlotUI> CrystalSlots;


    [Header("Staff")]
    public StaffSlotUI SlotStaffMain;
```

```
    public InventorySlotUI SlotStaffInventory;

    [Header("[== Hierarchy Data ==]")]
    public Transform Parent_Items;
    public Transform Parent_Floating;
    public Transform Parent_MainCrystal;

    [Header("[== Tool Tip Panel ==]")]
    public ToolTipPanel ToolTipPanel;

    public void Start()
    {
        Close();
        Init();
    }
    public void Init()
    {
        for (int i = 0; i < m_crystalHeight; i++)
        {
            AddCrystalRow();
        }

        foreach (var item in m_initItems)
        {
            PickUpItem(item);
        }
    }

    public void PickUpItem(Item _item)
    {
        Items.Add(_item);

        _item.Inventory = this;
        _item.OnPickUp();

        switch (_item.Type)
        {
            case ItemType.crystal:
                OnCrystalPickUp(_item);
                break;
            case ItemType.staff:
                OnStaffPickUp(_item);
                break;
        }
    }

    public void DropItem(Item _item)
    {
        Items.Remove(_item);

        _item.transform.position = player.ActiveActor != null ?
player.ActiveActor.transform.position : Vector3.zero;
        _item.OnDrop();
```

```
        _item.Inventory = null;
    }

    private void OnStaffPickUp(Item staffItem)
    {
        if (SlotStaffMain.AssignedItem == null)
        {
            staffItem.UIItem.AssignToSlot(SlotStaffMain);
            return;
        }

        if (SlotStaffInventory.AssignedItem == null)
        {
            staffItem.UIItem.AssignToSlot(SlotStaffInventory);
            return;
        }

        DropItem(SlotStaffInventory.AssignedItem.Item);
        staffItem.UIItem.AssignToSlot(SlotStaffInventory);

        return;
    }

    private void OnCrystalPickUp(Item crystalItem)
    {
        foreach (var slot in CrystalSlots)
        {
            if (slot.AssignedItem == null)
            {
                crystalItem.UIItem.AssignToSlot(slot);
                return;
            }
        }

        int newSlotIndex = CrystalSlots.Count;

        AddCrystalRow();

crystalItem.UIItem.AssignToSlot(CrystalSlots[newSlotIndex]);
    }

    private void OnCrystalSlotUpdated()
    {
        DeleteExtraCrystalRows();
    }

    private void AddCrystalRow()
    {
        for (int i = 0; i < m_crystalWidth; i++)
        {
            InventorySlotUI slot =
Instantiate(m_slotPrefabCrystal, m_crystalGrid.transform);
            CrystalSlots.Add(slot);
```

```
        }
    }

    private void DeleteExtraCrystalRows()
    {
        int overflowIndex = m_crystalWidth * m_crystalHeight;

        int slotsLeft = CrystalSlots.Count - overflowIndex;

        int RowsLeft = slotsLeft / m_crystalWidth;
        List<InventorySlotUI> slotsToDestroy = new();

        for (int i = 0; i < RowsLeft; i++)
        {
            bool thisRowHasItem = false;
            List<InventorySlotUI> thisRowSlots = new();

            for (int j = overflowIndex + i * m_crystalWidth; j <
overflowIndex + (i + 1) * m_crystalWidth; j++)
            {
                thisRowSlots.Add(CrystalSlots[j]);
                if (CrystalSlots[j].AssignedItem != null)
thisRowHasItem = true;
            }

            if (!thisRowHasItem)
            {
                slotsToDestroy.AddRange(thisRowSlots);
            }
        }

        foreach (var slot in slotsToDestroy)
        {
            CrystalSlots.Remove(slot);
            Destroy(slot.gameObject);
        }
    }

    public void Open()
    {
        m_InventoryAnimator.SetTrigger("Open");
    }

    public void Close()
    {
        m_InventoryAnimator.SetTrigger("Close");
    }

    public void OnGUI()
    {
        if (GUI.Button(new Rect(10, 90, 100, 30), "Open
Inventory"))
        {
```

```
            Open();
        }
        if (GUI.Button(new Rect(10, 130, 100, 30), "Close
Inventory"))
        {
            Close();
        }
    }
}
```

Б.2.14 Файл GameActor.cs

```
using UnityEngine;

public class GameActor : BaseBehaviour
{
    public ActorStats Stats = new();

    public Transform weaponHolder;

    public Vector2 actorDirection;
    public Transform focusTransform;

    public ActorDamageEffects actorDamageEffects;

    public bool isPlayer = false;

    public virtual float actorRotation { get; set; }
    public bool disableRotation = false;

    public float actorSpeed
    {
        get
        {
            if (navMeshAgent != null) return
navMeshAgent.velocity.magnitude;
            if (rb != null) return rb.velocity.magnitude;
            return 0f;
        }
    }

    private void Awake()
    {
        actorDamageEffects = new(this);
        Stats.Init(this);

        if (navMeshAgent != null)
        {
            navMeshAgent.updateRotation = false;
            navMeshAgent.updateUpAxis = false;
        }
    }
```

```
    private void LateUpdate()
    {
        UpdateAnimation();
    }

    public virtual void UpdateAnimation()
    {
        if (!disableRotation)
        {
            if (actorRotation < 45 && actorRotation > -45)
            {
                animator.SetInteger("LookDirection", 1);
                spriteRenderer.flipX = false;
            }
            if (actorRotation < 135 && actorRotation > 45)
            {
                animator.SetInteger("LookDirection", 4);
                spriteRenderer.flipX = false;
            }
            if (actorRotation > -135 && actorRotation < -45)
            {
                animator.SetInteger("LookDirection", 2);
                spriteRenderer.flipX = false;
            }
            if (actorRotation > 135 || actorRotation < -135)
            {
                animator.SetInteger("LookDirection", 3);
                spriteRenderer.flipX = true;
            }
        }

        if (actorSpeed > 0.09f) animator.SetBool("IsRunning",
true); else animator.SetBool("IsRunning", false);
    }
}
```

Б.2.15 Файл BehaviorController.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BehaviorController : BaseBehaviour
{
    private SortedDictionary<BehaviorType, List<BehaviorBase>>
behaviorPool = new();
    private Dictionary<BehaviorType, float> delaysByType =
new();
    private Dictionary<BehaviorType, float> lastExecutionTime =
new();

    private Coroutine currentCoroutine;
```

```
    private BehaviorBase currentBehavior;


    public GameActor target;
    public Room assignedRoom;

    public void Awake()
    {
        delaysByType.Add(BehaviorType.none, 0f);
        delaysByType.Add(BehaviorType.idle, 0f);
        delaysByType.Add(BehaviorType.approach, 0f);
        delaysByType.Add(BehaviorType.attack, 5f);

        lastExecutionTime.Add(BehaviorType.none, 0);
        lastExecutionTime.Add(BehaviorType.idle, 0);
        lastExecutionTime.Add(BehaviorType.approach, 0);
        lastExecutionTime.Add(BehaviorType.attack, 0);

        AddBehavior(new GenericBehaviors.GenericApproach(1.5f));
        AddBehavior(new GenericBehaviors.GenericWander(2f, 5f));
        AddBehavior(new GenericBehaviors.GenericIdle());
        AddBehavior(new
GenericBehaviors.WitchDoctorSuperSpiralAttack(7f, 3.5f, 1f));
        AddBehavior(new
GenericBehaviors.WitchDoctorSpiralAttack(7f, 3.5f, 1f));
        AddBehavior(new
GenericBehaviors.WitchDoctorCircleAttack(7f, 5f, 1f));
    }

    public void AddBehavior(BehaviorBase behavior)
    {
        if (!behaviorPool.ContainsKey(behavior.Type))
behaviorPool.Add(behavior.Type, new());
        behaviorPool[behavior.Type].Add(behavior);
    }

    public void Update()
    {
        Tick();
    }
    public void Tick()
    {
        BehaviorBase bestBehavior = null;

        foreach (var pair in behaviorPool)
        {
            if (lastExecutionTime[pair.Key] +
delaysByType[pair.Key] > Time.time) continue;

            List<BehaviorBase> bestBehaviors = new();
            foreach (var behavior in pair.Value)
            {
                if (!behavior.CanExecute(gameActor)) continue;
```

```
                if (currentBehavior != null &&
!currentBehavior.CanInterrupt()) continue;

                bestBehaviors.Add(behavior);
            }
            if (bestBehaviors.Count > 0) bestBehavior =
bestBehaviors[UnityEngine.Random.Range(0, bestBehaviors.Count)];
        }

        if (bestBehavior != null && bestBehavior !=
currentBehavior)
        {
            StartBehavior(bestBehavior);
        }
    }


    private void StartBehavior(BehaviorBase behavior)
    {
        StopCurrentBehavior();

        currentBehavior = behavior;
        currentBehavior.LastExecutionTime = Time.time;
        currentCoroutine =
gameActor.StartCoroutine(RunBehavior(currentBehavior));
        //Debug.Log($"Started behavior : {currentBehavior}");
    }

    private IEnumerator RunBehavior(BehaviorBase behavior)
    {
        yield return behavior.Execute(gameActor);
        lastExecutionTime[currentBehavior.Type] = Time.time;
        currentBehavior = null;
        currentCoroutine = null;
    }

    public void StopCurrentBehavior()
    {
        if (currentCoroutine != null)
        {
            gameActor.StopCoroutine(currentCoroutine);
            currentBehavior.OnStop(gameActor);
            //Debug.Log($"Stopped behavior :
{currentBehavior}");

            lastExecutionTime[currentBehavior.Type] = Time.time;
            currentCoroutine = null;
            currentBehavior = null;
        }
    }

    public bool IsRunning => currentCoroutine != null;
```

```
    public BehaviorBase CurrentBehavior => currentBehavior;
}
```

Б.2.16 Файл BulletManager.cs

```
using System;
using System.Collections.Generic;
using UnityEngine;

public class BulletManager : BaseBehaviour
{
    [NonSerialized]
    public List<Projectile> Projectiles = new();

    public ProjectileBank ProjectileBank;

    [NonSerialized]
    public float TimeScale = 1;

    public void Awake()
    {
        ProjectileBank.Initialize();
    }


    public Projectile CreateProjectile(string _bankName, Bullet
_bullet, float _damage, DamageType _damageType, Vector2
_position, float _speed, float _direction)
    {
        Projectile prefab = ProjectileBank.GetByName(_bankName);
        Projectile projectile = Instantiate(prefab, new
Vector3(_position.x, _position.y, 0), prefab.Rotatable ?
Quaternion.Euler(0, 0, _direction) : Quaternion.identity);

        Projectiles.Add(projectile);

        projectile.BulletManager = this;
        projectile.Speed = _speed;
        projectile.Damage = _damage;
        projectile.Bullet = _bullet;

        projectile.DamageType = _damageType;
        projectile.Direction = AngleToVector(_direction);

        projectile.Initialize();
        return projectile;
    }

    public Projectile CreateProjectile(string _bankName, Bullet
_bullet, float _damage, DamageType _damageType, Vector2
_position, float _speed, Vector2 _direction)
    {
        Projectile prefab = ProjectileBank.GetByName(_bankName);
```

```
        Projectile projectile = Instantiate(prefab, new
Vector3(_position.x, _position.y, 0), prefab.Rotatable ?
Quaternion.Euler(0, 0, Mathf.Atan2(_direction.y, _direction.x) *
Mathf.Rad2Deg) : Quaternion.identity);

        Projectiles.Add(projectile);

        projectile.BulletManager = this;
        projectile.Speed = _speed;
        projectile.Damage = _damage;
        projectile.Bullet = _bullet;

        projectile.DamageType = _damageType;
        projectile.Direction = _direction;

        projectile.Initialize();
        return projectile;
    }

    public Projectile CreateProjectile(string _bankName, Vector2
_position, float _speed, float _direction)
    {
        return CreateProjectile(_bankName, null, 0,
DamageType.none, _position, _speed, _direction);
    }

    public Projectile CreateProjectile(string _bankName,
DamageType _damageType, Vector2 _position, float _speed, float
_direction)
    {
        return CreateProjectile(_bankName, null, 0, _damageType,
_position, _speed, _direction);
    }



    public void DestroyProjectile(Projectile _projectile)
    {
        if (!Projectiles.Contains(_projectile)) return;
        Projectiles.Remove(_projectile);

        Destroy(_projectile.gameObject);
    }

    public static Vector2 AngleToVector(float angle)
    {
        float f = angle * (Mathf.PI / 180f);
        return new(Mathf.Cos(f), Mathf.Sin(f));
    }

    public static float VectorToAngle(Vector2 direction)
    {
```

```
        float angle = Mathf.Atan2(direction.normalized.y,
direction.normalized.x) * Mathf.Rad2Deg;
        return angle;
    }
}
```

Б.2.17 Файл Projectile.cs

```
using System;
using UnityEngine;

public class Projectile : BaseBehaviour
{
    public BulletManager BulletManager;
    [NonSerialized] private Vector2 m_direction = new(0, 0);
    [NonSerialized] public float Speed;

    [NonSerialized] public float Damage;

    [NonSerialized] public Vector2 Velocity;

    [NonSerialized] public Bullet Bullet;

    public bool damagesPlayer = false;
    public bool damagesEnemies = false;

    public float lifeTime = 5f;



    public bool Rotatable = true;

    [SerializeField]
    public bool overrideMovement;

    public DamageType DamageType
    {
        get => _damageType;
        set
        {
            _damageType = value;
            UpdateModel();
        }
    }
    public DamageType _damageType;

    public Vector2 Direction
    {
        get => m_direction;
        set
        {
            m_direction = value.normalized;
        }
```

```
}
private bool isDead = false;
public void Initialize()
{
    if (Bullet != null)
    {
        Bullet.Initialize(this);
        overrideMovement = Bullet.overrideMovement;
    }
}


private void Update()
{
    if (isDead) return;
    lifeTime -= Time.deltaTime;

    if (lifeTime <= 0) Die();

    if (!overrideMovement)
    {
        UpdateVelocity();
        UpdatePosition();
    }
    else
    {
        Bullet.UpdateVelocity();
        Bullet.UpdatePosition();
    }
}

protected virtual void UpdateVelocity()
{
    Velocity.x = Direction.x * Speed;
    Velocity.y = Direction.y * Speed;
}

protected virtual void UpdatePosition()
{
    Vector2 position = transform.localPosition;
    position.x += Velocity.x * Time.deltaTime;
    position.y += Velocity.y * Time.deltaTime;
    transform.localPosition = position;
}

public void UpdateModel()
{
    damageModel.TypeCondition |= DamageType;
}

public void Die()
{
    isDead = true;
```

```
        damageModel.OnDie();
        if (bulletManager != null)
BulletManager.DestroyProjectile(this);
        else Destroy(gameObject);
    }
}
```

Б.2.18 Файл Map.cs

```
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.UI;

public class Map : MonoBehaviour
{
    public Animator animator;
    public GameObject playerIcon;
    private void Awake()
    {
        animator = GetComponent<Animator>();
    }

    public void Open()
    {
        animator.SetTrigger("Open");
    }

    public void Close()
    {
        animator.SetTrigger("Close");
    }

    public void OnGUI()
    {
        if (GUI.Button(new Rect(10, 10, 100, 30), "Open Map"))
        {
            Open();
        }
        if (GUI.Button(new Rect(10, 50, 100, 30), "Close Map"))
        {
            Close();
        }
    }

    public Transform roomsTransform;
    public Vector2 roomPixelSize = new Vector2(5, 5);
    public Color roomColor = Color.white;

    public void Setup()
    {
        List<Room> rooms = FindObjectsOfType<Room>().ToList();
```

```
        foreach (Room room in rooms)
        {
            Texture2D texture = room.CreateRoomMinimapTexture();
            Sprite sprite = Sprite.Create(texture, new Rect(0,
0, texture.width, texture.height),
                new Vector2(0.5f, 0.5f), 1f, 0,
SpriteMeshType.FullRect);

            GameObject roomImageGO = new GameObject("RoomMap_" +
room.name, typeof(RectTransform), typeof(CanvasRenderer),
typeof(Image));
            roomImageGO.transform.SetParent(roomsTransform,
false);

            Image image = roomImageGO.GetComponent<Image>();
            image.sprite = sprite;
            image.color = roomColor;
            image.preserveAspect = true;

            RectTransform rect =
roomImageGO.GetComponent<RectTransform>();
            rect.pivot = new Vector2(0.5f, 0.5f); // Fix
alignment
            rect.sizeDelta = new Vector2(texture.width *
roomPixelSize.x, texture.height * roomPixelSize.y);

            Vector3 worldCenter = room.GetRoomCenterWorld();
            rect.anchoredPosition =
WorldToMinimapPosition(worldCenter);

            room.mapImage = roomImageGO;
            roomImageGO.SetActive(false);
        }
    }

    public Transform player; // Assign in inspector or find at
runtime
    private Vector2 originalRoomTransformPos;

    private void Start()
    {
        originalRoomTransformPos = roomsTransform.localPosition;
    }

    private void LateUpdate()
    {
        if (player != null && roomsTransform != null)
        {
            Vector2 offset = new Vector2(
                -player.position.x * minimapScale.x,
                -player.position.y * minimapScale.y
            );
```

```
            roomsTransform.localPosition =
originalRoomTransformPos + offset;
        }
    }


    public Vector2 minimapScale = new Vector2(5f, 5f); // UI px
per world unit
    public Vector2 minimapOriginOffset = Vector2.zero; // Offset
if your rooms are negative

    Vector2 WorldToMinimapPosition(Vector3 worldPos)
    {
        return new Vector2(
            worldPos.x * minimapScale.x,
            worldPos.y * minimapScale.y
        ) + minimapOriginOffset;
    }

}
```

Б.2.19 Файл FogOfWarParticleSystem.cs

```
using UnityEngine;
using System.Collections.Generic;

public class FogOfWarParticleSystem : MonoBehaviour
{
    [Header("Particle Settings")]
    public int particleCount = 500;
    public float particleSize = 1f;
    public float noiseSpeed = 0.5f;
    public float boxHalfSize_x = 30f;
    public float boxHalfSize_y = 15f;
    public float fadeSpeed = 2f;

    [Header("Positioning")]
    public float maxDepth = 6;
    public float minDepth = .5f;

    [Header("Rendering")]
    public Gradient colorGradient;
    public Mesh particleMesh;
    public Material particleMaterial;

    [Header("Atlas Settings")]
    public int atlasCols = 2;
    public int atlasRows = 2;
    private Vector2 uvCellSize;

    [Header("Camera Follow")]
    public Transform cameraTransform;
```

```
    private struct Particle
    {
        public Vector3 position;
        public Vector3 noiseOffset;
        public float baseColor;
        public float alpha;

        public Vector2 uvScale;
        public Vector2 uvOffset;
        public bool wasWrappedOnThisFrame;
        public float fadeSpeedFactor;
    }

    private List<Particle> particles;
    private MaterialPropertyBlock propBlock ;
    private List<Matrix4x4> matrices;
    private List<Vector4> colors;

    private void Awake()
    {
        particles = new();
        propBlock = new();
        matrices = new();
        colors = new();


        uvCellSize = new Vector2(1f / atlasCols, 1f /
atlasRows);
    }

    private void Start()
    {
        if (cameraTransform == null) cameraTransform =
Camera.main.transform;
        if (particleMesh == null)
            particleMesh = CreateQuad();

        for (int i = 0; i < particleCount; i++)
        {
            int index = Random.Range(0, atlasCols * atlasRows);
            int col = index % atlasCols;
            int row = index / atlasCols;

            Vector2 uvOffset = new Vector2(col * uvCellSize.x,
row * uvCellSize.y);

            Vector3 randPos = cameraTransform.position + new
Vector3(
                Random.Range(-boxHalfSize_x, boxHalfSize_x),
                Random.Range(-boxHalfSize_y, boxHalfSize_y),
                Random.Range(minDepth, maxDepth)
            );
```

```
            Particle p = new Particle
            {
                position = randPos,
                noiseOffset = new Vector3(Random.value * 10f,
Random.value * 10f, 0f),
                baseColor = Random.value,
                alpha = 1f,
                uvScale = uvCellSize,
                uvOffset = uvOffset,
                wasWrappedOnThisFrame = false,
                fadeSpeedFactor = Random.Range(-0.3f, 0.3f)
            };

            particles.Add(p);
        }
    }

    private void Update()
    {
        Vector3 cameraPos = cameraTransform.position;
        float sizeX = boxHalfSize_x * 2f;
        float sizeY = boxHalfSize_y * 2f;

        for (int i = 0; i < particles.Count; i++)
        {
            Particle p = particles[i];
            p.wasWrappedOnThisFrame = false;

            // Noise movement
            float dx = Mathf.PerlinNoise(p.noiseOffset.x,
Time.time * noiseSpeed) - 0.5f;
            float dy = Mathf.PerlinNoise(p.noiseOffset.y,
Time.time * noiseSpeed) - 0.5f;
            p.position += new Vector3(dx, dy, 0f) *
Time.deltaTime;

            // Wrap around camera box
            Vector3 relative = p.position - cameraPos;

            Vector3 originalRelative = relative;

            relative.x = Mathf.Repeat(relative.x +
boxHalfSize_x, sizeX) - boxHalfSize_x;
            relative.y = Mathf.Repeat(relative.y +
boxHalfSize_y, sizeY) - boxHalfSize_y;

            p.wasWrappedOnThisFrame =
!Mathf.Approximately(relative.x, originalRelative.x) ||
!Mathf.Approximately(relative.y, originalRelative.y);

            p.position = cameraPos + relative;

            // Fade
```

```
            bool revealed = Physics2D.OverlapPoint(p.position,
LayerMask.GetMask("FogRevealer")) != null;
            float targetAlpha = revealed ? 0f : 1f;

            if (p.wasWrappedOnThisFrame)
                p.alpha = targetAlpha;
            else
                p.alpha = Mathf.MoveTowards(p.alpha,
targetAlpha, (fadeSpeed + p.fadeSpeedFactor) * Time.deltaTime);

            particles[i] = p;
        }
    }

    private void LateUpdate()
    {
        matrices.Clear();
        colors.Clear();

        List<Vector4> uvOffsets = new();
        List<Vector4> uvScales = new();

        foreach (var p in particles)
        {
            if (p.alpha <= 0f) continue;

            matrices.Add(Matrix4x4.TRS(p.position,
Quaternion.identity, Vector3.one * particleSize));
            Color finalColor =
colorGradient.Evaluate(p.baseColor);
            finalColor.a *= p.alpha;
            colors.Add(finalColor);

            uvOffsets.Add(p.uvOffset);
            uvScales.Add(p.uvScale);
        }

        // Render
        for (int i = 0; i < matrices.Count; i += 1023)
        {
            int count = Mathf.Min(1023, matrices.Count - i);

            propBlock.Clear();
            propBlock.SetVectorArray("_BaseColor",
colors.GetRange(i, count));
            propBlock.SetVectorArray("_UVOffset", uvOffsets);
            propBlock.SetVectorArray("_UVScale", uvScales);


            Graphics.DrawMeshInstanced(particleMesh, 0,
particleMaterial, matrices.GetRange(i, count), propBlock);
        }
    }
```

```
    private Mesh CreateQuad()
    {
        var mesh = new Mesh();
        mesh.vertices = new Vector3[] {
            new(-0.5f, -0.5f, 0),
            new(0.5f, -0.5f, 0),
            new(-0.5f,  0.5f, 0),
            new(0.5f,  0.5f, 0),
        };
        mesh.uv = new Vector2[] {
            new(0, 0), new(1, 0),
            new(0, 1), new(1, 1),
        };
        mesh.triangles = new int[] { 0, 2, 1, 2, 3, 1 };
        mesh.RecalculateNormals();
        return mesh;
    }
}
```